C/C++ Contributing Editors

The New C: Variable Length Arrays, Part 2

Randy Meyers

Unlike C or C++, C99 lets you define the bounds of multidimensional arrays at run time, much to scientific programmers' delight.

In my last column [1], I discussed the deficiencies of arrays in C before C99. The definitions of pointer arithmetic and the index operator in C intertwine the concepts of pointer and array. Before C99, C required that the size of objects be a compile-time constant. Since pointer arithmetic and thus array indexing depend on the size of objects, this restriction made arrays (particularly multidimensional arrays) less flexible in C than other languages.

C99 removed the restriction that the size of arrays needs to be a compile-time constant by allowing run-time expressions to be used as the bounds of an array. Arrays with run-time bounds are called VLAs (variable length arrays). VLAs have three major benefits:

1. The size of arrays (even multidimensional arrays) can be appropriate for the problem at hand, even if that size cannot be known until run time.
2. The C implementation automatically provides storage management for VLAs.
3. Functions or other code that processes multidimensional arrays can be more flexible. The bounds of any dimension of the array can be passed as an argument rather than fixed at compile time.

Some C programmers might be surprised that the third benefit above is lacking in pre-C99 C. The reason for this is that single-dimensional arrays are more common than multidimensional arrays in the traditional application areas of C (systems programming, application programming, embedded programming). C has always been able to handle arrays whose first (leftmost) or only dimension was not known because the size of the total array itself as opposed to the size of its elements (which might be sub-arrays in a multidimensional array) is not needed to do the pointer arithmetic underlying the index operator. However, if you have more than a single dimension and the size of the second or later dimensions was not known at compile time, then arrays in C before C99 became almost useless. Unfortunately, this situation frequently occurs in numerical programming (“you have N equations with M variables...”). My previous column [1] explains this problem in more detail. While I believe that column will make you appreciate VLAs more, this month’s column is understandable without that background.

VLAs

As we will see in my next column, the new VLA feature affects not only arrays, but also pointers, function arguments and parameters, and even **typedefs**. However, in this column, we will look at arrays of variable length themselves.

If an array has a run-time expression as its bounds, then it is a VLA. If an array has a constant expression [2] (an expression that can be evaluated at compile time) as its bounds or empty braces (only permitted in a few contexts), then the array is not a VLA and has the same semantics such arrays have always had in C.

The bounds of a VLA must have integer type and must evaluate to a value greater than zero. If the bounds is less than one, then the program has a run-time error that the C implementation has no obligation to catch: the program might terminate immediately, continue to run with mysterious behavior, or even appear to work. The bounds expression of a VLA can use any operator, even assignment and function call. However, if the bounds expression is a comma expression, it must be enclosed in parentheses.

VLAs must be auto (as opposed to static or extern) variables in a block. When the block containing the array is entered and the declaration of the array is reached, the implementation evaluates the bounds expressions and allocates the array with that bounds. When the array goes out of scope, the C implementation automatically deallocates it.

```
#define BOUNDS(a) ((sizeof (a))/(sizeof ((a)[0])))

void example1(int n)
{
   double array[n][n+1];

   printf("sizeof array = %u\n",
      (unsigned) sizeof array);
   printf("1st dimension = %u\n",
      (unsigned) BOUNDS(array));
   printf("2nd dimension = %u\n",
      (unsigned) BOUNDS(array[0]));
}
```

In the above example, **array** is a VLA since at least one of its dimensions is a run-time expression (in fact, both bounds are). When **example 1** is called and control reaches the declaration of **array**, the implementation will evaluate the bounds expressions, calculate the sizes of the dimensions of the array, and save those results for later use. It will then allocate space for **array**. If the argument **n** to the function had the value **3**, then **array** will act as if it had been declared as:

```
double array[3][4];
```

Typical implementations will allocate the space for the array on the stack. This is a very efficient operation since it only involves adding or subtracting a value from the stack pointer. However, at least one implementation stores VLAs on a heap and generates function calls to allocate and deallocate VLAs.

Unlike non-variable length arrays, **sizeof** is a run-time operation on VLAs. The C implementation will use the bounds information it saves when it allocates the array to compute **sizeof** when needed. If you assume that **sizeof** (**double**) is **8** and that the parameter **n** has the value **3**, then **example1(3)** prints:

```
sizeof array = 96
1st dimension = 3
2nd dimension = 4
```

The **BOUNDS** macro is a C idiom used by many programmers to compute the bounds of an array by dividing the size of the array by the size of its element. The **BOUNDS** macro works for any array, whether variable length or not, and uses no C99-specific features. If the array is not variable length, then **BOUNDS** is a compile-time constant expression. If the array is variable length, then **BOUNDS** is a run-time expression. **BOUNDS** can calculate the bounds of any dimension of a multidimensional array. If **x** is a multidimensional array, then **BOUNDS(x)** is the first dimension, **BOUNDS(x[0])** is the second dimension, **BOUNDS(x[0][0])** is the third dimension, etc. In general, it is more efficient for you to save your own copy of the bounds of a VLA, but if you fail to do so, you can use the **BOUNDS** macro to calculate the bounds based on the C implementation's saved information.

The size and bounds of a VLA are fixed from the point the declaration of the array is executed until the array goes out of scope.

```
void example2(int n)
{
   double array[n][n+1];
   n += 10;
   printf("sizeof array = %u\n",
      (unsigned) sizeof array);
   printf("1st dimension = %u\n",
      (unsigned) BOUNDS(array));
   printf("2nd dimension = %u\n",
      (unsigned) BOUNDS(array[0]));
}
```

In **example2**, the size of **array** does not change even though the variable originally used to calculate its dimensions has a new value. The call **example2(3)** prints the same results as **example1(3)**. The C implementation uses the information it saves when the array is declared whenever it needs to know the size of the array or any of its dimensions. It does not reevaluate the original bounds expressions.

There is a sequence point at the end of a full declarator of an object in a declaration, but there are no sequence points associated with nested array declarators. This means that a compiler is free to evaluate the bounds expressions of a multidimensional VLA in any order it chooses, but all of the bounds expressions of one VLA must complete their evaluations before beginning the evaluations of bounds expressions of the next object. For example,

```
int a1[f()][g()][h()], a2[k()];
```

The compiler might call **f**, **g**, and **h** in any order, but it must have called all three before calling **k**.

When a VLA goes out of scope or its lifetime ends, the VLA is deallocated. Two more common ways for this to happen are by returning from a function or by exiting the block containing the VLA. However, C99 permits mixing statements and declarations [2], and the lifetime of an object in C99 ends if a **goto** branches backwards to a label before the object's declaration [2].

```
void example3()
{
   int i = 1;
   char a1[i];
   ++i;
loop:
   char a2[i];
   ++i;
   printf("sizeof a1 = %u\n",
      (unsigned) sizeof a1);
   printf("sizeof a2 = %u\n",
      (unsigned) sizeof a2);
   printf("i = %d\n", i);
   if (i < 4)
     goto loop;
   printf("last sizeof a2 = %u\n",
      (unsigned) sizeof a2);
}
```

In **example3**, **goto loop** causes **a2**'s lifetime to end immediately before branching to **loop**. In contrast, the lifetimes of **a1** and **i** do not end since their declarations appear before the label **loop**. After branching to **loop**, the declaration of **a2** is executed for the second time with the latest value for **i**. This causes this program to print the following:

```
sizeof a1 = 1
sizeof a2 = 2
i = 3
sizeof a1 = 1
sizeof a2 = 3
i = 4
last sizeof a2 = 3
```

Note that the dynamic ending of an object's lifetime does not mean that it is out of scope and cannot be referenced by later code. In **example3**, the lifetime of **a2** only ends if the **goto** is executed. If the **goto** is not executed, the lifetime does not end, and since **a2** is still in scope, it may legitimately be used by later code. **a2** has the size and contents that it had before the **if** statement skipped over the **goto**. (Interestingly, C++ constructors and destructors behave exactly this same way: a backwards **goto** causes an automatic object's destructor to execute, and the object will be constructed again when its declaration is executed.)

Limitations on VLAs

- Since a VLA must be an automatic variable of a block, a VLA cannot be declared at file scope.
- A VLA cannot have static or extern storage class.
- A VLA cannot be a member of a struct or union.
- You cannot initialize a VLA.
- A compound literal [3] cannot have a VLA type.
- A **goto** or a switch statement cannot branch into the scope of a VLA from outside the scope. (Such a branch would skip over the code that allocates the storage for the VLA.)

- **longjmp** may cause storage to be lost if you **longjmp** out of a block (directly or indirectly) that contains currently allocated VLAs. As was stated above, at least one C implementation allocates VLAs on a heap rather than on the stack. Normally, when you **longjmp** out of a function, all automatic variables of blocks exited by the **longjmp** are deallocated because **longjmp** resets the stack pointer to the earlier value saved by **setjmp**. However, resetting the stack pointer does nothing to deallocate any objects allocated on a separate heap, and thus those allocations might never be deallocated. The first implementation of VLAs (Cray), which was also the basis of the VLA features in C99, had this problem with **longjmp**. I suspect that most implementations will allocate VLAs on the stack and not have this problem. However, for maximum portability if this storage leak would be an issue for your programs, avoid mixing VLAs and **longjmp**.

Next Month

Next month's column will cover pointers to VLAs and VLAs as function parameters (a special case of pointers to VLAs).

References

[1] Randy Meyers. "The New C: Why Variable Length Arrays?," *C/C++ Users Journal*, October 2001.

[2] Randy Meyers. "The New C: Declarations and Initializations," *C/C++ Users Journal*, April 2001, <www.cuj.com/reference/articles/2001/0104/0104d/0104d.htm>.

[3] Randy Meyers. "The New C: Compound Literals," *C/C++ Users Journal*, June 2001.

Randy Meyers *is a consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*